Theoretical Computer Science 85 (1991) 53–74
Elsevier

Parallel parsing on a one-way linear array of finite-state machines*

Oscar H. Ibarra, Tao Jiang** and Hui Wang

Department of Computer Science, University of California, Santa Barbara, CA93106, USA

Communicated by A. Salomaa
Received January 1989
Revised July 1989

Abstract

Ibarra, O.H., T. Jiang and H. Wang, Parallel parsing on a one-way linear array of finite-state machines, Theoretical Computer Science 85 (1991) 53–74.

Efficient parallel algorithms for some parsing problems are presented. These problems include the parsing of linear context-free languages, languages accepted by nondeterministic one-counter automata, and transductions defined by a special class of two-tape nondeterministic finite-state transducers. The model of parallel computation is a one-way linear array of identical finite-state machines. The data movement in the array is one-way, from left to right. For inputs of length n, the array uses n nodes. Our algorithms can actually produce a parse, i.e. a sequence of rules (moves) that generates (accepts) an input, in linear time. When only a no/yes answer is required, the parsing problem becomes a recognition problem. The best serial (RAM) algorithms for the corresponding recognition problems take $O(n^2/\log^2 n)$ time and space. Previous parallel algorithms for the recognition problems run in linear time on a one-way linear array of finite-state machines.

1. Introduction

The main motivation for this paper comes from the following problem: Given a grammar (acceptor) G, develop an efficient algorithm which for any input x outputs "no" if G does not generate (accept) x; otherwise, the algorithm outputs a parse, i.e. a sequence of rules (moves) that generates (accepts) x. We call this a "parsing" problem. If we require the algorithm only to output "no" or "yes" (instead of a parse), then we have a "recognition" problem. In general, parsing is more difficult than recognition.

It is well known that parsing of context-free languages (CFL's) can be implemented on a multitape Turing machine (actually, a transducer which can output at most one

* This research was supported in part by NSF Grants CCR-8918409 and CCR-9096221.

** Present affiliation: Department of Computer Science, McMaster University, Hamilton, Ontario, Canada L8S 4K1.

symbol per move) in $O(n^3)$ time and $O(n^2)$ space [14]. Here we consider two important subclasses of context-free languages: linear context-free languages (LCFL's) and languages accepted by nondeterministic one-counter automata. It was shown in [3] that recognition of LCFL's can be done on a multitape Turing machine in $O(n^2)$ time and $O(n)$ space, but that parsing takes $O(n^2)$ time and $O(n^2)$ space. The reason for the extra space needed in parsing is that in the dynamic programming method for generating a parse, the $O(n^2)$ entries of a two-dimensional recognition matrix (which is constructed to determine if the string is generated by the grammar) are needed to recover a parse. The recognition matrix is evaluated row by row, and each row is dependent only on the previous row. Since for recognition only the last row is needed, the space used for the computation is only $O(n)$. It was left unanswered in [3] whether the space requirement of $O(n^2)$ for parsing can be reduced (without increasing the time). We show that this can be done. In fact, we show that parsing of LCFL's can be done on a *single-tape* Turing machine (*without* a separate read-only input tape) in $O(n^2)$ time and $O(n)$ space. The same result applies to parsing of nondeterministic pushdown automata which make only one-turn on their pushdown stack. We also look at nondeterministic one-counter automata (1-NCA's). (These are pushdown automata with unary stack alphabet.) It is known that recognition can be done on a single-tape Turing machine in $O(n^2)$ time and $O(n)$ space [4]. We show that the same time and space bounds hold for parsing. The proof in this case is slightly more complicated because these machines have counters which not only use unbounded memory but also make unbounded number of counter turns. We also consider the parsing problems for other devices such as two-tape (or two-head) nondeterministic finite-state transducers and show that they too can be solved on a single-tape Turing machine in $O(n^2)$ time and $O(n)$ space.

Our results are actually stronger. We define a one-way linear iterative array (OLIA, for short) as a finite one-dimensional array of identical finite-state machines (nodes) in which information is allowed to move only in one direction: from left to right. For an input sequence of length n, the array uses n nodes which are initially set to the quiescent state. The serial input is applied to the leftmost node and the serial output is observed from the rightmost node. Arrays which allow only one-way communication have nice properties, e.g. with respect to problem decomposition and fault-tolerance [11, 12]. Thus, OLIA's are especially attractive for VLSI implementation. Clearly, an OLIA operating in linear time can be simulated by a single-tape Turing machine in $O(n^2)$ time and $O(n)$ space. In this paper, we show that the parsing problems mentioned above can be implemented on an OLIA in linear time. The main proof technique is a divide-and-conquer (recursive) strategy, which we show can be implemented on an OLIA. As far as we know, this is the first paper that exhibits an implementation of recursion on a one-way (or two-way) linear array.

The paper is organized as follows. Section 2 formally defines the OLIA. Section 3 gives an equivalent formulation of an OLIA in terms of a restricted type of sequential machine. All the constructions in the paper are given in terms of this

equivalent formulation. Section 4 presents a linear-time algorithm (on an OLIA) for the parsing of transductions defined by discrete two-tape nondeterministic finite-state transducers (2-TNFT's). Discrete 2-TNFT's were introduced in [13] as generalizations of two-tape (or two-head) finite automata. They have applications in the study of parallel processes [13]. An interesting corollary is that parsing of discrete 2-TNFT transductions can be done on a single-tape Turing machine in quadratic time and linear space. Section 5 shows that the LCFL parsing problem can be recast in terms of the parsing of discrete 2-TNFT transductions. It follows that LCFL parsing can be implemented on an OLIA in $O(n)$ time, and hence, also on a single-tape Turing machine in $O(n^2)$ time and $O(n)$ space. Section 6 considers the parsing problem for 1-NCA's and shows that this problem can also be solved on an OLIA in $O(n)$ time.

2. One-way linear iterative array

Figure 1 shows a one-way linear iterative array (OLIA) with serial input/output and one-way communication between nodes. Each node is a finite-state machine. The array has n identical nodes for inputs of length n. The nodes operate synchronously at discrete time steps by means of a common clock (not shown in the figure). The input $a_1 a_2 \ldots a_n\$$ is fed serially to the leftmost node, while the serial output $b_1 b_2 \ldots b_k\$$ is observed from the rightmost node. The a_i's (b_i's) are symbols from some fixed finite input (output) alphabet. Both input and output sequences are terminated by a special delimiter \$. (We assume that \$ is not in the input and output alphabets.) The leftmost node receives a_i, $1 \leqslant i \leqslant n$, at time $i-1$, and \$ after time $n-1$. Hence, unlike the a_i's, \$ is not "consumed" when read by the OLIA, and is always available for rereading. The state and output of a node at time t are functions of its state and the state of its left neighbor (or the input in the case of the leftmost node) at time $t-1$. Thus, $S_i^{t+1} = f(S_{i-1}^t, S_i^t)$ and $O_i^{t+1} = g(S_{i-1}^t, S_i^t)$ for some functions f and g, where S_i^t and O_i^t denote the state and output of the ith node at time t. We assume that $S_0^t = a_{t+1}$ for $0 \leqslant t < n$ and $S_0^t = \$$ for $t \geqslant n$. We require that $S_i^0 = q_0$ (the quiescent state), $O_i^0 = \lambda$ (representing blank), $f(q_0, q_0) = q_0$ and $g(q_0, q_0) = \lambda$. (Thus, at time 0, each node is in the quiescent state q_0, with its output set to λ. It remains in state q_0 with output λ until its left neighbor enters a nonquiescent state.) The serial output $b_1 \ldots b_k\$$ is observed from the rightmost node starting at time n. Thus, the outputs at times $1, 2, \ldots, n-1$, which are λ's, are not included in $b_1 \ldots b_k\$$. The OLIA has time complexity $T(n)$ on input $a_1 \ldots a_n\$$ if it outputs \$ after at most $T(n)$ time. Clearly, $T(n) \geqslant 2n$.

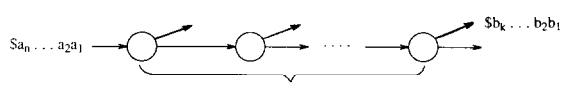

Fig. 1. An OLIA.

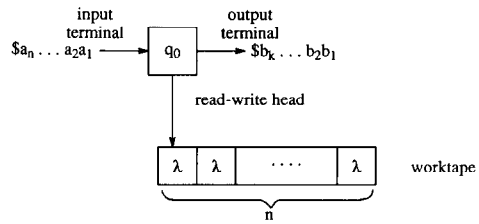

Fig. 2. An SM.

3. Describing the algorithms

There is a nice way to represent the computation of an OLIA in terms of its time–space diagram (or unrolling). The representation can be described in terms of a restricted type of sequential machine called SM. An SM (see Fig. 2) has only one finite-state processor with an input terminal which receives the serial input $a_1 \ldots a_n\$$ and an output terminal from which the serial output $b_1 \ldots b_k\$$ is observed. As in an OLIA, we assume that \$ is not consumed when read by the machine and is always available for rereading. The SM operates on a one-dimensional worktape of n cells for inputs of length n. Each cell can hold a symbol from some fixed finite alphabet. Initially, all cells of the worktape are set to λ (representing blank).

The SM operates in sweeps as shown in Fig. 3. A sweep begins with the processor of the machine in a quiescent state q_0 and the read–write head (RWH) scanning the leftmost cell. The machine then reads an input and scans the worktape from left to right. For each cell scanned, the machine rewrites the cell, produces an output, and changes state. After processing the rightmost cell, the RWH is reset to the leftmost cell in state q_0. Then it begins the next sweep. The outputs $b_1, \ldots, b_k, \$$ are the outputs of the rightmost cell on sweeps $1, \ldots, k+1$. The SM outputs \$ only after reading the input delimiter. The SM has sweep complexity $S(n)$ on input $a_1 \ldots a_n\$$ if it outputs \$ after at most $S(n)$ sweeps. Clearly, $S(n) \geqslant n+1$.

sweep	input	worktape			observed output
0		λ	λ	λ	λ
1	a_1	Z^1_1	Z^1_2	Z^1_3	b_1
2	a_2	Z^2_1	Z^2_2	Z^2_3	b_2
3	a_3	Z^3_1	Z^3_2	Z^3_3	b_3
4	\$	Z^4_1	Z^4_2	Z^4_3	b_4
5	\$	Z^5_1	Z^5_2	Z^5_3	\$

Fig. 3. The worktape profile of an SM on input $a_1 a_2 a_3\$$.

The relationship between an OLIA and an SM is given by the following theorem which has essentially been proved in [9] (see also [7]). For completeness, we include a short proof.

Theorem 3.1. *Let $S(n) \geq n+1$. An OLIA A operating in $S(n)+n-1$ time can be simulated by an SM M in $S(n)$ sweeps, and vice versa. Moreover, the conversion from an SM to an equivalent OLIA (and vice versa) can be carried out automatically and efficiently.*

Proof. We only illustrate the constructions by examples. If we "unroll" the computation of A in time and space, we obtain the trellis structure C_1 of combinational cells as shown in Fig. 4 for input $a_1a_2a_3\$$. We use only the portion of C_1 above the line $a_1q_1^1q_2^1q_3^1$, since the portion below the line represents the cells of A while still in the quiescent state. (The λ's appearing along the vertical lines in C_1 represent the quiescent state of the cells of A. The figure shows the outputs only of the rightmost

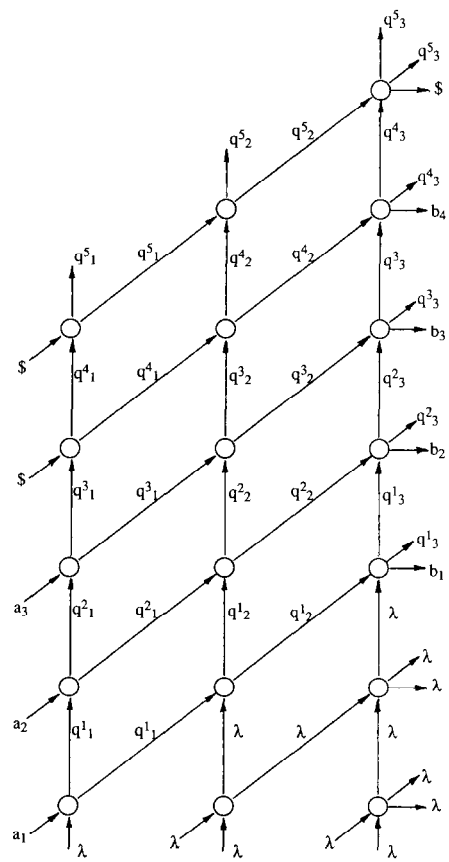

Fig. 4. Trellis C_1.

sweep	input	worktape			observed output
0		λ	λ	λ	λ
1	a_1	q^1_1	q^1_2	q^1_3	b_1
2	a_2	q^2_1	q^2_2	q^2_3	b_2
3	a_3	q^3_1	q^3_2	q^3_3	b_3
4	\$	q^4_1	q^4_2	q^4_3	b_4
5	\$	q^5_1	q^5_2	q^5_3	S

Fig. 5. The worktape profile of an SM simulating C_1.

cell.) We can now construct an SM M simulating C_1. M essentially "simulates" the diagonals of C_1. The worktape profile of M is shown in Fig. 5. The contents of the worktape on sweep $i+1$ can easily be obtained from the contents of sweep i. For example, when M scans q_1^2 on reading input symbol a_3, it replaces the scanned symbol by q_1^3, outputs a symbol and enters state q_1^3. When in this state, M scans symbol q_2^2, it replaces the scanned symbol by q_2^3, outputs a symbol, and enters state q_2^3. On scanning symbol q_3^2, M rewrites it by q_3^3, outputs b_3, and enters state q_3^3. Then, it is reset to the leftmost cell in state q_0.

Figure 3 shows the worktape profile of an SM M on input $a_1a_2a_3\$$ and the observed outputs. To see how an OLIA A can simulate M, we transform the profile into a trellis C_2 as shown in Fig. 6. In the figure, q_i^j is the state of M after writing symbol Z_i^j (see Fig. 3). Clearly, C_2 is the unrolling of the desired OLIA A.

One can easily verify that A has time complexity $S(n)+n-1$ if and only if M has sweep complexity $S(n)$. □

Since an SM is a sequential machine, it is relatively easier to program than an OLIA: we do not have to deal with the problems of concurrency and synchronization present in an OLIA. Thus, one can develop an efficient SM algorithm and then convert it into an OLIA algorithm. Note that by Theorem 1, the OLIA algorithm is efficient if and only if the equivalent SM algorithm is efficient.

4. Parsing of discrete 2-TNFT transductions

A two-tape nondeterministic finite-state transducer (2-TNFT) $M=(Q, \Sigma, \Delta, \delta, q_0, F)$ is a nondeterministic finite automaton with 2 one-way input tapes (one head per tape) and 1 one-way output tape, where

- Q is the state set,
- Σ is the input alphabet,
- Δ is the output alphabet,
- $\delta: Q\times(\Sigma\cup\varepsilon)\times(\Sigma\cup\varepsilon)\to 2^{Q\times\Delta^*}$ is the set of *moves*,

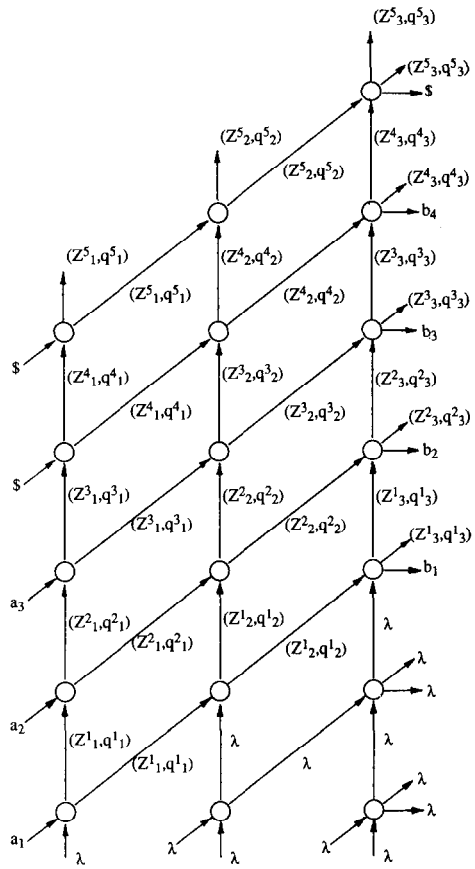

Fig. 6. Trellis C_2 simulating the SM M.

- q_0 is the initial state, and
- $F \subseteq Q$ is the set of accepting states.

The move $(q, a, b) \to (p, c)$ means: if M is in state q with the two input heads reading a and b respectively, $a, b \in \Sigma \cup \{\varepsilon\}$, then M changes its state to p, input head 1 moves $|a|$ cells to the right, input head 2 moves $|b|$ cells to the right, and M outputs string c. We assume without loss of generality that at least one input head moves to the right every step. Note that, if a (b) is ε, then input head 1 (head 2) is stationary. M defines a set of transductions $T(M) = \{(x_1, x_2, y) \mid M$ on input (x_1, x_2) can output y and enter an accepting state$\}$. M is called *discrete* if it outputs exactly one symbol whenever one of the two input heads moves to the right [13]. When the two input heads move to the right at the same time, M outputs two symbols. Hence, if (x_1, x_2, y) is in $T(M)$, then $|y| = |x_1| + |x_2|$. The transductions defined by a discrete 2-TNFT are called discrete 2-TNFT transductions. For simplicity, we consider only the transductions that satisfy $|y| = |x_1| + |x_2|$.

Let M be a fixed 2-TNFT and $t=(x_1, x_2, y)$ be a transduction in $T(M)$. A *parse* of t is a sequence of moves that takes M on input (x_1, x_2) to an accepting state with output y. The 2-TNFT transduction parsing problem is stated as follows: Given a transduction $t=(x_1, x_2, y)$, output a parse of t if $t \in T(M)$, "no" otherwise.

Discrete 2-TNFT transductions are quite useful. They have applications, e.g. in parallel processes [13]. In Section 5, we will show that the LCFL parsing problem can be recast in terms of the parsing of discrete 2-TNFT transductions. It was shown in [13] that discrete 2-TNFT transductions can be recognized on a RAM in $O((n+m)^2/\log(n+m))$ time and space, where n and m are the lengths of x_1 and x_2. The time and space were improved to $O((n+m)^2/\log^2(n+m))$ in [8]. A linear-time OLIA algorithm for recognizing discrete 2-TNFT transductions was given in [10]. We will show that discrete 2-TNFT transductions can be parsed on an OLIA in $O(n)$ time.

Let $M=(Q, \Sigma, \Delta, \delta, q_0, F)$ be a fixed discrete 2-TNFT. Let $t=(x_1, x_2, y)=(a_1 \ldots a_n, b_1 \ldots b_m, c_1 \ldots c_{n+m})$ be a transduction. We can check if t is in $T(M)$ as follows [13, 10]. Denote the configuration when M is in state q and input heads 1 and 2 are at positions i and j (i.e. after reading $a_1 \ldots a_i$ and $b_1 \ldots b_j$), respectively, by (q, i, j), where $q \in Q$, $0 \leqslant i \leqslant n$ and $0 \leqslant j \leqslant m$. The initial configuration is $(q_0, 0, 0)$. Construct an $(n+1) \times (m+1)$ matrix H such that $H(i,j)=\{q \mid M$ can enter (q, i, j) and output $c_1 \ldots c_{i+j}\}$, $0 \leqslant i \leqslant n$ and $0 \leqslant j \leqslant m$. Hence, (x_1, x_2, y) is in $T(M)$ if and only if $H(n, m)$ contains an accepting state. The matrix H is called the *recognition matrix* of M on transduction (x_1, x_2, y). Matrix H can be evaluated using the following recurrence equations:

$$
\begin{aligned}
&H(0,0)=\{q_0\};\\
&H(0,j)=\{q \mid \exists p \text{ in } H(1,j-1) \text{ such that } (p, \varepsilon, b_j) \to (q, c_j) \text{ is in } \delta\}, \qquad 0<j \leqslant m;\\
&H(i,0)=\{q \mid \exists p \text{ in } H(i-1,1) \text{ such that } (p, a_i, \varepsilon) \to (q, c_i) \text{ is in } \delta\}, \qquad 0<i \leqslant n;\\
&H(i,j)=\{q \mid \exists p \text{ in } H(i-1,j) \text{ such that } (p, a_i, \varepsilon) \to (q, c_{i+j}) \text{ is in } \delta\}\\
&\qquad \cup \{q \mid \exists p \text{ in } H(i,j-1) \text{ such that } (p, \varepsilon, b_j) \to (q, c_{i+j}) \text{ is in } \delta\}\\
&\qquad \cup \{q \mid \exists p \text{ in } H(i-1,j-1) \text{ such that } (p, a_i, b_j) \to (q, c_{i+j-1} c_{i+j}) \text{ is in } \delta\},\\
&\qquad\qquad 0<i \leqslant n,\ 0<j \leqslant m.
\end{aligned}
$$

The recognition matrix H can be evaluated row by row (column by column), starting from the first row (the first column). Clearly, row (column) $i+1$ depends only on row (column) i. Given row (column) i, the entries in row (column) $i+1$ can be easily obtained one by one, starting from $H(i+1, 0)$ ($H(0, i+1)$), using the recurrence equations.

We shall show how to parse the transductions defined by M on an SM in $O(n+m)$ sweeps (and hence, on an OLIA in $O(n+m)$ time). First, we present a serial (RAM) algorithm that can parse a transduction in $O(nm)$ time and $O(n+m)$ space. The basic idea behind the algorithm is divide-and-conquer. A similar technique was used in [5] for solving the longest common subsequence problem.

Without loss of generality, we assume the discrete 2-TNFT M has only one accepting state, i.e. $F=\{f\}$. We will denote the "reverse" of M by M^R, i.e. $M^R=(Q, \Sigma, \Delta, \delta^R, f, \{q_0\})$, where $\delta^R=\{(q, a, b) \to (p, c) \mid (p, a, b) \to (q, c)$ in $\delta\}$. For any q, p in Q, let $M_{q,p}=(Q, \Sigma, \Delta, \delta, q, \{p\})$ and $M^R_{q,p}=(Q, \Sigma, \Delta, \delta^R, q, \{p\})$. Let

lastrow(H, q, x_1, x_2, y) (lastrow(H^R, q, x_1, x_2, y)) denote the function that returns the last row of the recognition matrix H of $M_{q,f}$ (M^R_{q,q_0}) on transduction (x_1, x_2, y) and lastcolumn(H, q, x_1, x_2, y) (lastcolumn(H^R, q, x_1, x_2, y)) denote the function that returns the last column of the recognition matrix H of $M_{q,f}$ (M^R_{q,q_0}) on (x_1, x_2, y). Note that the recognition matrix of a 2-TNFT on a transduction does not depend on the accepting states of the 2-TNFT. It is easy to see that these functions can be computed in $O(nm)$ time and $O(n+m)$ space, where $n=|x_1|$ and $m=|x_2|$. The following algorithm can parse a transduction in $O(nm)$ time and $O(n+m)$ space. A call TPARSE(q, p, x_1, x_2, y, S) to the algorithm will find a parse, with respect to $M_{q,p}$, of the transduction (x_1, x_2, y). The parse will be stored in S. In particular, TPARSE(q_0, f, x_1, x_2, y, S) gives a parse (with respect to M) of transduction (x_1, x_2, y).

Algorithm TPARSE(q, p, x_1, x_2, y, S);
begin
{Let $x_1 = a_1 \ldots a_n$, $x_2 = b_1 \ldots b_m$, and $y = c_1 \ldots c_{n+m}$.
For convenience, we assume $n+m>0$}
1. {If $n, m \leqslant 1$ then solve the problem directly}
if $n=1$ and $m=0$ **then**
if $(q, a_1, \varepsilon) \rightarrow (p, c_1)$ is in δ **then** $S := (q, a_1, \varepsilon) \rightarrow (p, c_1)$
else $S :=$ "no"
elseif $n=0$ and $m=1$ **then**
if $(q, \varepsilon, b_1) \rightarrow (p, c_1)$ is in δ **then** $S := (q, \varepsilon, b_1) \rightarrow (p, c_1)$
else $S :=$ "no"
elseif $n=1$ and $m=1$ **then**
if $(q, a_1, b_1) \rightarrow (p, c_1 c_2)$ is in δ **then** $S := (q, a_1, b_1) \rightarrow (p, c_1 c_2)$
elseif $\exists u \in Q$ such that $(q, a_1, \varepsilon) \rightarrow (u, c_1)$ and $(u, \varepsilon, b_1) \rightarrow (p, c_2)$ are in δ **then**
$S := (q, a_1, \varepsilon) \rightarrow (u, c_1)$, $(u, \varepsilon, b_1) \rightarrow (p, c_2)$
elseif $\exists u \in Q$ such that $(q, \varepsilon, b_1) \rightarrow (u, c_1)$ and $(u, a_1, \varepsilon) \rightarrow (p, c_2)$ are in δ **then**
$S := (q, \varepsilon, b_1) \rightarrow (u, c_1)$, $(u, a_1, \varepsilon) \rightarrow (p, c_2)$
else $S :=$ "no"
2. **else** {Split the problem}
2.1. **if** $n \geqslant m$ **then**
2.1.1. $i := \lceil n/2 \rceil$;
row1 $:=$ lastrow$(H, q, a_1 \ldots a_i, b_1 \ldots b_m, c_1 \ldots c_{i+m})$;
row2 $:=$ lastrow$(H^R, p, a_n \ldots a_{i+1}, b_m \ldots b_1, c_{n+m} \ldots c_{i+1})$;
2.1.2. {Find a j such that $\exists u \in Q$, $(a_1 \ldots a_i, b_1 \ldots b_j, c_1 \ldots c_{i+j})$ is in $T(M_{q,u})$
and $(a_{i+1} \ldots a_n, b_{j+1} \ldots b_m, c_{i+j+1} \ldots c_{n+m})$ is in $T(M_{u,p})$}
$j := 0$;
while row1$(j) \cap$ row2$(m-j) = \emptyset$ **do**
$j := j+1$;
end;
if $j > m$ **then** $S :=$ "no"; **return endif**
$u :=$ an arbitrary state in row1$(j) \cap$ row2$(m-j)$

2.2. **else** $\{n<m\}$
2.2.1. $j:=\lceil m/2\rceil$;
column1 := lastcolumn$(H, q, a_1 \ldots a_n, b_1 \ldots b_j, c_1 \ldots c_{n+j})$;
column2 := lastcolumn$(H^R, p, a_n \ldots a_1, b_m \ldots b_{j+1}, c_{n+m} \ldots c_{j+1})$;
2.2.2. {Find an i such that $\exists u \in Q$, $(a_1 \ldots a_i, b_1 \ldots b_j, c_1 \ldots c_{i+j})$ is in $T(M_{q,u})$
and $(a_{i+1} \ldots a_n, b_{j+1} \ldots b_m, c_{i+j+1} \ldots c_{n+m})$ is in $T(M_{u,p})$}
$i:=0$;
while column1$(i) \cap$column2$(n-i) = \emptyset$ **do**
$i:=i+1$;
end;
if $i>n$ **then** S:="no"; **return endif**
u:=an arbitrary state in column1$(i) \cap$column2$(n-i)$
endif;
3. {Solve simpler problems}
call TPARSE$(q, u, a_1 \ldots a_i, b_1 \ldots b_j, c_1 \ldots c_{i+j}, S_1)$;
call TPARSE$(u, p, a_{i+1} \ldots a_n, b_{j+1} \ldots b_m, c_{i+j+1} \ldots c_{n+m}, S_2)$;
4. $S:=S_1, S_2$
endif
end.

The algorithm TPARSE works as follows. To find a parse, with respect to $M_{q,p}$, of a transduction (x_1, x_2, y), TPARSE first checks if the lengths of both x_1 and x_2 are $\leqslant 1$. If yes, it finds a parse of (x_1, x_2, y) directly. Otherwise, it divides the transduction into two smaller transductions and finds a parse, with respect to some 2-TNFT, for each of them. Clearly, the space complexity of TPARSE is $O(n+m)$ and the time complexity of TPARSE is $O(nm)+O(nm/2)+O(nm/4)+\cdots=O(nm)$.

Theorem 4.1. *Parsing of discrete 2-TNFT transductions can be done on an OLIA in linear time.*

Proof. Let $t=(x_1, x_2, y)$ be a transduction. We show that the algorithm TPARSE can be carried out on an SM Z in $16(n+m)$ sweeps. Let $x_1=a_1 \ldots a_n$, $x_2=b_1 \ldots b_m$, and $y=c_1 \ldots c_{n+m}$. The input to Z is $a_1 \ldots \hat{a_n} b_1 \ldots \hat{b_m} c_1 \ldots c_{n+m}\$$ (the marker $\hat{}$ attached to the last symbols of x_1 and x_2 is used to indicate the boundaries between x_1 and x_2, and x_2 and y). The worktape is conceptually divided into four tracks. The tracks are numbered 1 through 4, from top to bottom. Each cell of the worktape is viewed as a 4×7 array of subcells. Thus, the worktape of Z on transduction t is a $4 \times 14(n+m)$ array of subcells. Z operates in three stages: input stage, parse stage and output stage. The input stage takes $2(n+m)$ sweeps. In this stage, Z simply reads the input string and places it in the first $2(n+m)$ subcells of track 1. The parse stage takes $11(n+m)$ sweeps. In this stage, Z executes TPARSE(q_0, f, x_1, x_2, S). When the algorithm TPARSE divides a problem into two subproblems, Z splits the problem and solves the two subproblems in parallel. At the end of this stage, a parse (with respect to M) of t is

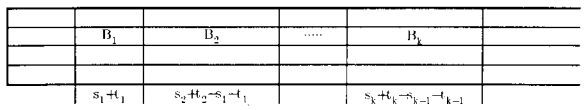

Fig. 7. The active blocks on Z's worktape.

stored on the worktape. The output stage takes $3(n+m)$ sweeps. In this stage, Z outputs the parse obtained in the parse stage. The parse is output in reverse order, i.e. the first move of M is output the last and the last move is output the first. In what follows, we sketch the operations of Z in the parse stage and leave most of the details to the reader.

At any time in the parse stage, the worktape contains a sequence of *active blocks*: $B_1, B_2, \ldots, B_k$, for some $1 \leqslant k \leqslant n+m$ (see Fig. 7). Each active block B_r represents a run of the algorithm TPARSE on states q_r and p_r, and transduction $(a_{s_{r-1}+1} \ldots a_{s_r}, b_{t_{r-1}+1} \ldots b_{t_r}, c_{s_{r-1}+t_{r-1}+1} \ldots c_{s_r+t_r})$, $1 \leqslant r \leqslant k$, where $0 = s_0 \leqslant s_1 \leqslant \cdots \leqslant s_k = n$ and $0 = t_0 \leqslant t_1 \leqslant \cdots \leqslant t_k = m$. Initially, there is only one active block, i.e. the block representing a run of the algorithm TPARSE on states q_0 and f, and transduction t. The length of block B_r is $2(s_r + t_r - s_{r-1} - t_{r-1})$ subcells, $1 \leqslant r \leqslant k$. When the algorithm TPARSE splits a problem $(q_r, p_r, a_{s_{r-1}+1} \ldots a_{s_r}, b_{t_{r-1}+1} \ldots b_{t_r}, c_{s_{r-1}+t_{r-1}+1} \ldots c_{s_r+t_r})$ (i.e. the parsing of transduction $(a_{s_{r-1}+1} \ldots a_{s_r}, b_{t_{r-1}+1} \ldots b_{t_r}, c_{s_{r-1}+t_{r-1}+1} \ldots c_{s_r+t_r})$ with respect to M_{q_r, p_r}) into two subproblems $(q_r, u, a_{s_{r-1}+1} \ldots a_i, b_{t_{r-1}+1} \ldots b_j, c_{s_{r-1}+t_{r-1}+1} \ldots c_{i+j})$ and $(u, p_r, a_{i+1} \ldots a_{s_r}, b_{j+1} \ldots b_{t_r}, c_{i+j+1} \ldots c_{s_r+t_r})$, the active block B_r is divided into two active blocks, one (the left block) for solving the problem $(q_r, u, a_{s_{r-1}+1} \ldots a_i, b_{t_{r-1}+1} \ldots b_j, c_{s_{r-1}+t_{r-1}+1} \ldots c_{i+j})$ and the other (the right block) for solving the problem $(u, p_r, a_{i+1} \ldots a_{s_r}, b_{j+1} \ldots b_{t_r}, c_{i+j+1} \ldots c_{s_r+t_r})$. Since Z makes left-to-right sweeps, it can pass data only from left to right (on its worktape). Later on, we will see that Z needs to move data (i.e. strings of symbols) from right to left in an active block. The right-to-left data movement in active blocks is achieved by shifting all the active blocks one subcell to the right every sweep. It is easy to see that, since the active blocks are shifted to a subcell to the right every sweep, Z can move a datum d subcells to the left in an active block in d sweeps.

We describe the operations of Z in an active block. Without loss of generality, consider the initial active block, i.e. the block for problem (q_0, f, x_1, x_2, y). Denote the block by B. The initial configuration of B is shown in Fig. 8(a). If $n \leqslant 1$ and $m \leqslant 1$, Z solves the problem directly as in step 1 of the algorithm TPARSE. Let S_0 be the parse obtained. Z stores S_0 in B. The active block B will not be changed in the following sweeps and the parse S_0 will be the output associated with the block B. Suppose $n > 1$ or $m > 1$. Then Z executes steps 2.1–2.2.2 of the algorithm TPARSE in three phases. Instead of deciding whether $n \geqslant m$ or $n < m$, first Z performs 2.1.1–2.1.2 (for the case $n \geqslant m$) and 2.2.1–2.2.2 (for the case $n < m$) in parallel. At the end of the first phase, Z will know which of $n \geqslant m$ or $n < m$ is true and discard the computation for the false case. Without loss of generality, we assume $n \geqslant m$ and describe the operations of

Z for this case. Each phase takes $n+m$ sweeps. In the sequel, all the data movements and locations of subcells are taken to be with respect to block B.

In the first phase, Z verifies that $n \geqslant m$, marks the ith symbol a_i of x_1 with a $^-$, where $i = \lceil n/2 \rceil$, moves x_2 to the subcells under $c_1 \ldots c_m$, and reverses x_1, x_2 and y as follows. The configuration of B at the end of the first phase is shown in Fig. 8(b). Z can keep track of the number of sweeps it has spent in the phase by propagating a marker from the last subcell to the first subcell of track 1, at unit speed (i.e. one subcell per sweep). Using the same idea, the symbol a_i can be located as follows. Z propagates a marker from the first subcell of track 1 to the right at unit speed. At the same time, it propagates another marker from the nth subcell to the left at unit speed. Clearly, the two markers meet at the symbol a_i. The shifting of x_2 to the subcells under $c_1 \ldots c_m$ is straightforward. The reverse of x_1 can be obtained as in reversing a string on a stack, i.e. Z takes the symbols $a_1, \ldots, a_n$ from the string, one symbol per sweep, and pushes them back into the subcells 1 through m on track 4. This is possible because Z can shift a copy of x_1 to the left at unit speed. The reverses of x_2 and y can be obtained similarly. To simplify the presentation, Z also keeps an extra copy of y under $a_1 \ldots \hat{a_n} b_1 \ldots b_m$.

In the second phase, Z first computes

$$\text{row1} = \text{lastrow}(H, q_0, a_1 \ldots a_i, b_1 \ldots b_m, c_1 \ldots c_{i+m})$$

and

$$\text{row2} = \text{lastrow}(H^R, a_n \ldots a_{i+1}, b_m \ldots b_1, c_{n+m} \ldots c_{i+1})$$

and stores them on tracks 2 and 4, one entry per subcell (see Fig. 8(c)). We only describe how row1 is obtained; row2 can be obtained similarly. Let H_1 be the recognition matrix of $M_{q_0,f}$ $(=M)$ on transduction $(a_1 \ldots a_i, b_1 \ldots b_m, c_1 \ldots c_{i+m})$. Z evaluates H_1 row by row, starting from row 0, as follows. It propagates a marker from the first subcell to the ith subcell of track 1 and shifts the string y (on track 1) to the left, all at unit speed. In the first sweep, Z computes row 0 of H_1 using $b_1 \ldots b_m$ and $c_1 \ldots c_m$ and overwrites the copy of x_2 by row 0. In the second sweep, Z remembers a_1 (in its state), computes row 1 using a_1, $c_1 \ldots c_m$ and row 0, and overwrites row 0 by row 1. In the third sweep, Z remembers a_2, computes row 2 using a_2, $c_1 \ldots c_{m+1}$ and row 1, and overwrites row 1 by row 2. Generally, in the kth sweep, $3 < k \leqslant i+1$, Z remembers a_{k-1}, computes row $k-1$ using a_{k-1}, $c_{k-2} \ldots c_{m+k-2}$ and row $k-2$, an overwrites row $k-2$ by row $k-1$. It takes $i+1$ sweeps to compute row1 and row2.

When row1 and row2 are obtained, Z reverses row2 in m sweeps. The configuration of B after the second phase is shown in Fig. 8(d) (row2^R denotes the reverse of row2).

In the third phase, Z first marks the entry row1(j) with a special symbol, where j is the smallest k satisfying row1$(k) \cap$ row$2^R(k) \neq \emptyset$, $0 \leqslant k \leqslant m$. This can be done in one sweep. Then Z moves the substring $a_{i+1} \ldots a_n$ and the extra copy of y towards the right. It is easy to see that, by using the markers in B, Z can rearrange the strings so that the

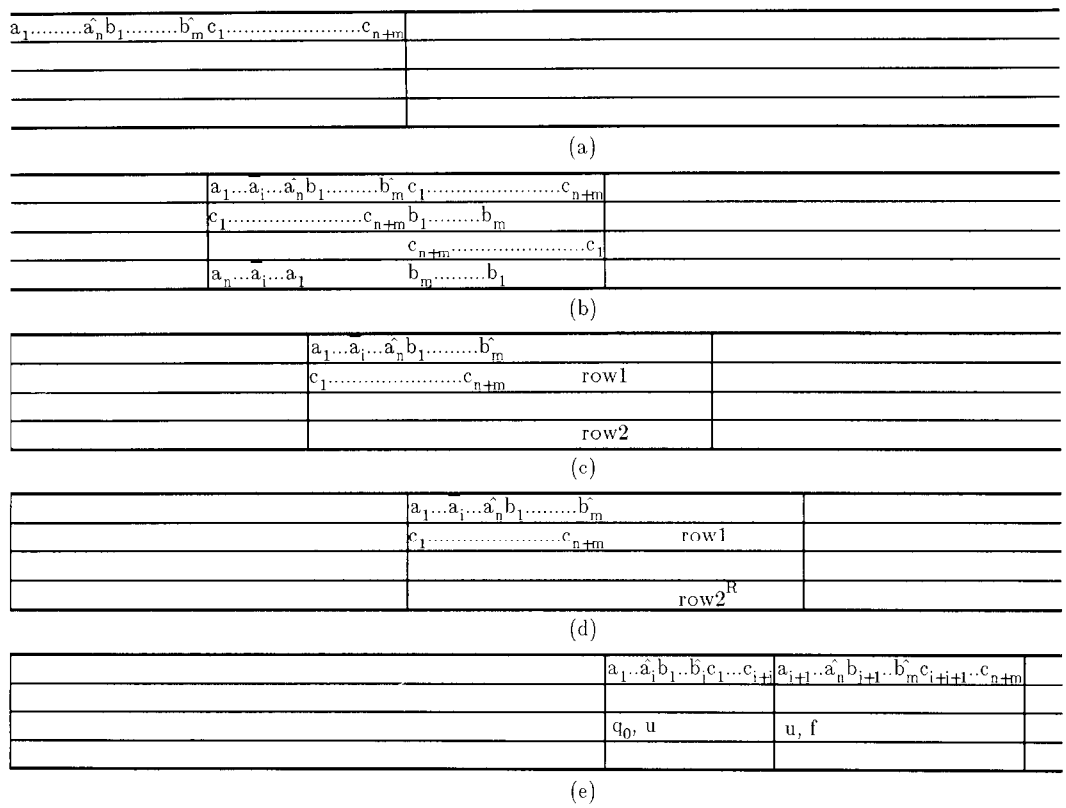

Fig. 8. The changes of the configuration of block B.

block configuration is as shown in Fig. 8(e). [In the figure, u is an arbitrary state in $\text{row1}(j) \cap \text{row2}^R(j)$]. When this is done, Z has divided B into two active blocks. The left block is of length $2(i+j)$ (subcells) and is used to solve the problem $(q_0, u, a_1 \ldots a_i, b_1 \ldots b_j, c_1 \ldots c_{i+j})$. The right block is of length $2(n+m-i-j)$ (subcells) and is used to solve the problem $(u, f, a_{i+1} \ldots a_n, b_{j+1} \ldots b_m, c_{i+j+1} \ldots c_{n+m})$. Note that, since B is the initial active block, the states q_0 and f were not stored in B at the beginning.

Let $N = 3(n+m)$. It is easy to see that the output stage takes at most $N + \frac{3}{4}N + \frac{1}{2}N + \frac{3}{8}N + \frac{1}{4}N + \frac{3}{16}N + \cdots = \frac{7}{4}(N + \frac{1}{2}N + \frac{1}{4}N + \cdots) = \frac{7}{2}N$ sweeps. Thus, the sweep complexity of Z is $2(n+m) + \frac{21}{2}(n+m) + 3(n+m) \leqslant 16(n+m)$. □

Corollary 4.2. *Parsing of discrete 2-TNFT transductions can be done on a single-tape Turing machine in quadratic time and linear space.*

As an application, we show that the string shuffling problem [13] can be solved on an OLIA in linear time. The problem is defined as follows: Given three strings x, y and z (over alphabet Σ), determine if z is a shuffle of x and y and if so, find one such shuffle (i.e. how z can be obtained as a shuffle of x and y). It was shown in [13] that the

set $L=\{(x,y,z)\mid z$ is a shuffle of x and $y\}$ can be recognized on a RAM in $O((n+m)^2/\log(n+m))$ time, where n and m are the lengths of x and y. In [10] it is shown that the set L can be recognized on an OLIA in linear time. The string shuffling problem (which is more than just the recognition of L) can be restated as a discrete 2-TNFT transduction parsing problem. We define a discrete 2-TNFT

$$M=(\{q_0\},\Sigma,\Sigma,\delta,q_0,\{q_0\}),$$

where

$$\delta=\{(q_0,a,\varepsilon)\to(q_0,a)\mid a\in\Sigma\}\cup\{(q_0,\varepsilon,a)\to(q_0,a)\mid a\in\Sigma\}.$$

It is easy to see that $T(M)=L$. Hence, given strings x, y and z, a parse of the transduction (x,y,z) tells us how z can be obtained as a shuffle of x and y.

Corollary 4.3. *The string shuffling problem can be solved on an OLIA in linear time.*

One can define a two-head nondeterministic finite-state transducer (2-HNFT) as a nondeterministic finite automaton with 2 independent heads on a single one-way input tape and 1 one-way output tape. A two-tape (two-head) nondeterministic finite automaton is a 2-TNFT (2-HNFT) without an output tape.

Corollary 4.4. *The parsing problem for 2-HNFT's, two-tape nondeterministic finite automata, and two-head nondeterministic finite automata are solvable on an OLIA in linear time.*

5. Parsing of LCFL's

An important subclass of context-free languages (CFL's) is the class of linear context-free languages (LCFL's). A context-free grammar $G=(V,\Sigma,P,S)$ is a linear context-free grammar (LCFG) if each rule in P is of the form $A\to uBv$ or $A\to u$, where $A,B\in V$ and $u,v\in\Sigma^*$. (V is the set of nonterminals, Σ is the set of terminal symbols, P is the set of rules, and S is the starting nonterminal.) The language generated by an LCFG is called an LCFL. Let $G=(V,\Sigma,P,S)$ be an LCFG. G is in *normal* form if the rules in P are of the form $A\to aB$, $A\to Ba$ or $A\to a$, where $a\in\Sigma$ (i.e. a is a single terminal symbol). It is easy to transform any LCFG which does not generate ε to one in normal form. Thus, without loss of generality, we will only consider normal LCFG's. Let x be a string in $L(G)$. A parse of x is a sequence of rules that derives x from S. The definition of the parsing problem for LCFL's is straightforward.

A RAM algorithm for LCFL recognition running in $O(n^2)$ time and $O(n)$ space was given in [3], but the question of whether or not a parse can be found in the same time and space was left unanswered, where n is the length of x. The time for LCFL recognition was improved to $O(n^2/\log^2 n)$ in [8]. It was also shown in [1, 6] that

LCFL recognition can be done on an OLIA in $O(n)$ time. Here we show that parsing of LCFL's can be done on an OLIA in $O(n)$ time.

Interestingly, the LCFL parsing problem can be recast in terms of the parsing of discrete 2-TNFT transductions. Let $G=\langle V, \Sigma, P, S\rangle$ be a normal LCFG. We define a discrete 2-TNFT $M=(Q, \Sigma, \Delta, \delta, S, \{f\})$ as follows:

$$
\begin{aligned}
&Q = V \cup \{f\} \quad (f \notin V \text{ is a new symbol}),\\
&\Delta = \{0, 1\},\\
&\delta = \{(A, a, \varepsilon) \to (B, 0) \mid A \to aB \in P\}\\
&\quad \cup \{(A, \varepsilon, a) \to (B, 0) \mid A \to Ba \in P\}\\
&\quad \cup \{(A, a, \varepsilon) \to (f, 0) \mid A \to a \in P\}\\
&\quad \cup \{(f, a, \varepsilon) \to (f, 1) \mid a \in \Sigma\}\\
&\quad \cup \{(f, \varepsilon, a) \to (f, 1) \mid a \in \Sigma\}.
\end{aligned}
$$

Note that, in a computation, exactly one input head of M moves per step. Clearly, for any string x, x is in $L(G)$ if and only if the transduction $(x, x^R, 0^{|x|}1^{|x|})$ is in $T(M)$. (x^R denotes the reverse of x.) Suppose x is a string in $L(G)$. Let $W=w_1, w_2, \ldots, w_{2n}$ be a parse of transduction $(x, x^R, 0^{|x|}1^{|x|})$, where $n=|x|$, $w_i=(q_i, a_i, b_i)\to(q_{i+1}, 0)$, $1 \leqslant i \leqslant n$, $q_1=S$, $q_{n+1}=f$, and $w_i=(f, a_i, b_i)\to(f, 1)$, $n+1 \leqslant i \leqslant 2n$. Define $r_i=q_i \to a_i q_{i+1}$ if $a_i \neq \varepsilon$, $q_i \to q_{i+1} b_i$ if $b_i \neq \varepsilon$, $1 \leqslant i \leqslant n-1$, and $r_n=q_n \to a_n$. Then, it is easy to see that $R=r_1, r_2, \ldots, r_n$ is a parse of string x. Hence the following corollary.

Corollary 5.1. *LCFL's can be parsed on an OLIA in linear time.*

The following corollary improves the result in [3].

Corollary 5.2. *LCFL's can be parsed on a single-tape Turing machine in* $O(n^2)$ *time and* $O(n)$ *space.*

6. Parsing of 1-NCA languages

Another important subclass of CFL's is the class of languages accepted by nondeterministic one-counter automata (1-NCA's). A 1-NCA is a nondeterministic finite automaton with a one-way input tape and a counter. A 1-NCA M is denoted by a 5-tuple $(Q, \Sigma, \delta, q_0, F\}$, where Q is the state set, Σ is the input alphabet, $\delta: Q \times (\Sigma \cup \varepsilon) \times \{0, 1\} \to 2^{Q \times \{-1, 0, 1\}}$ is the set of moves, q_0 is the initial state, and $F \subseteq Q$ is the set of accepting states. The move $(q, a, e)\to(p, d)$ means: if M is in state q with input head reading a (in $\Sigma \cup \varepsilon$) and counter status e (0 for empty counter, 1 otherwise), M changes its state to p, input head moves one cell to the right, and adds value d to the counter. (Note that M is not allowed to make a decrement in an empty counter.) The language accepted by M, denoted by $L(M)$, is $\{x \mid x \in \Sigma^*$ and M on input x can enter an accepting state$\}$. It has been shown in [2] that 1-NCA's are equivalent to real-time 1-NCA's. Thus, without loss of generality, we assume that M advances its input head

on each atomic move and accepts with the counter empty [2]. Again, we assume that F contains only one accepting state f. Let x be a string. A parse of x (with respect to M) is a sequence of moves that takes M on input x to the accepting state f with the counter empty. Note that in an accepting computation the counter value of M is always $\leqslant n$.

It was shown in [4] that 1-NCA languages can be recognized on a RAM in $O(n^2)$ time, where n is the length of x. Again, the time was improved to $O(n^2/\log^2 n)$ in [8]. It was shown in [7] that the recognition of 1-NCA languages can be done on an OLIA in $O(n)$ time. Using the techniques given in Section 4, we can show that 1-NCA languages can be parsed on an OLIA in $O(n)$ time.

Let $M=(Q, \Sigma, \delta, q_0, \{f\})$ be a fixed 1-NCA. Let q_1, q_2 be any two states in Q and c_1, c_2 be nonnegative integers. Define a generalized 1-NCA M_{q_1,c_1,q_2,c_2} (a variation of M) as follows. The state set and move function of M_{q_1,c_1,q_2,c_2} are Q and δ, respectively. Given any input, M_{q_1,c_1,q_2,c_2} starts the computation with state q_1 and counter value c_1. An input x is accepted by M_{q_1,c_1,q_2,c_2} if M_{q_1,c_1,q_2,c_2} can enter state q_2 and counter value c_2 after reading x. We can check if a string $x=a_1 \ldots a_n$ is accepted by M_{q_1,c_1,q_2,c_2} as follows. Since M_{q_1,c_1,q_2,c_2} can make a decrement (or increment) in the counter by at most n during an accepting computation on input x, we assume $|c_2-c_1| \leqslant n$. Let $b_1=\max\{0, \lceil (c_1+c_2-n)/2 \rceil\}$ and $b_2=\lfloor (c_1+c_2+n)/2 \rfloor$. Construct an $(n+1)\times(b_2-b_1+1)$ matrix E such that $E(i,j)=\{q \mid M_{q_1,c_1,q_2,c_2}$ can enter state q with counter value j after reading $a_1 \ldots a_i\}$, $0 \leqslant i \leqslant n$ and $b_1 \leqslant j \leqslant b_2$. Thus, x is accepted by M_{q_1,c_1,q_2,c_2} if and only if $E(n, c_2)$ contains the state q_2. E is called the recognition matrix of M_{q_1,c_1,q_2,c_2} on x. The recognition matrix E can be evaluated using the following recurrence equations [4, 8]. For any nonnegative integer c, let $\text{pos}(c)=1$ if $c>0$, 0 if $c=0$.

$$
\begin{aligned}
&E(0, c_1)=\{q_1\},\\
&E(0, j)=\emptyset, \quad b_1 \leqslant j \leqslant b_2,\ j \neq c_1,\\
&E(i, j)=\{q \mid j>b_1 \text{ and } \exists p \text{ in } E(i-1, j-1) \text{ such that}\\
&\qquad (p, a_i, \text{pos}(j-1)) \to (q, 1) \text{ is in } \delta\}\\
&\qquad \cup\{q \mid \exists p \text{ in } E(i-1, j) \text{ such that } (p, a_i, \text{pos}(j)) \to (q, 0) \text{ is in } \delta\}\\
&\qquad \cup\{q \mid j<b_2 \text{ and } \exists p \text{ in } E(i-1, j+1) \text{ such that } (p, a_i, 1) \to (q, -1) \text{ is in } \delta\},\\
&\qquad\qquad 1 \leqslant i \leqslant n,\ b_1 \leqslant j \leqslant b_2.
\end{aligned}
$$

Clearly, the matrix E can be evaluated row by row, starting from the first row.

Before we can describe the parsing algorithm, we need to define another matrix E^{R}. Define $E^{\mathrm{R}}(i, j)=\{q \mid$ starting from state q and counter value j, M_{q_1,c_1,q_2,c_2} can reach state q_2 with counter value c_2 after reading $a_i \ldots a_1\}$, $0 \leqslant i \leqslant n$, $b_1 \leqslant j \leqslant b_2$. E^{R} is called the *reverse recognition matrix* of M_{q_1,c_1,q_2,c_2} on x^{R}. Matrix E^{R} can be computed according to the following recurrence equations.

$$
\begin{aligned}
&E^{\mathrm{R}}(0, c_2)=\{q_2\},\\
&E^{\mathrm{R}}(0, j)=\emptyset, \quad b_1 \leqslant j \leqslant b_2,\ j \neq c_2,
\end{aligned}
$$

$$E^R(i,j)=\{q \mid j>b_1 \text{ and } \exists p \text{ in } E^R(i-1,j-1) \text{ such that } (q,a_i,1)\to(p,-1) \text{ is in } \delta\}$$
$$\cup\{q \mid \exists p \text{ in } E^R(i-1,j) \text{ such that } (q,a_i,\text{pos}(j))\to(p,0) \text{ is in } \delta\}$$
$$\cup\{q \mid j<b_2 \text{ and } \exists p \text{ in } E^R(i-1,j+1) \text{ such that } (q,a_i,\text{pos}(j))\to(p,1) \text{ is in } \delta\}, \quad 1\leqslant i\leqslant n,\ b_1\leqslant j\leqslant b_2.$$

Let lastrow(E,q_1,c_1,x) (lastrow(E^R,q_2,c_2,x^R)) denote the function that returns the last row of recognition matrix E (the reverse recognition matrix E^R) of M_{q_1,c_1,q_2,c_2} on string x. Clearly, the function lastrow can be computed on a RAM in $O(n^2)$ time and $O(n)$ space.

We present a RAM algorithm CPARSE for parsing $L(M)$. The call CPARSE(q_1,c_1,q_2,c_2,x,S) to the algorithm gives a parse (with respect to M_{q_1,c_1,q_2,c_2}) of string x. The parse will be stored in S. A parse (with respect to M) of the input string x can be found by calling CPARSE$(q_0,0,f,0,x,S)$.

Algorithm CPARSE(q_1,c_1,q_2,c_2,x,S);
begin
{Let $x=a_1\ldots a_n$. For simplicity, assume $n>0$.
b_1 and b_2 are defined as in the above discussion}
1. {If $n=1$ then solve the problem directly}
if $n=1$ **then**
if $(q_1,a_1,\text{pos}(c_1))\to(q_2,c_2-c_1)\in\delta$ **then** $S:=(q_1,a_1,\text{pos}(c_1))\to(q_2,c_2-c_1)$
else $S:=$"no"
2. **else** {Split the problem}
2.1. $i:=\lceil n/2\rceil$;
row1 $:=$ lastrow$(E,q_1,c_1,a_1\ldots a_i)$;
row2 $:=$ lastrow$(E^R,q_2,c_2,a_n\ldots a_{i+1})$;
2.2. {Find a j such that $\exists p\in Q$, $p\in$row1$(j)\cap$row2(j)}
$j:=b_1$;
while row1$(j)\cap$row2$(j)=\emptyset$ **do**
$j:=j+1$
end;
if $j>b_2$ **then** $S:=$"no"; **return**
else $p:=$an arbitrary state in row1$(j)\cap$row2(j);
3. {Solve simpler problems}
call CPARSE$(q_1,c_1,p,j,a_1\ldots a_i,S_1)$;
call CPARSE$(p,j,q_2,c_2,a_{i+1}\ldots a_n,S_2)$;
4. $S:=S_1,S_2$
endif
end.

It is easy to see that the space complexity of CPARSE is $O(n)$ and the time complexity of CPARSE is $O(n^2)$.

There is a problem that hinders the implementation of the algorithm CPARSE on an SM. Let $x=a_1...a_n$ be an imput to M. During a computation on x, the value of M's counter may be of the same magnitude as n. Since each cell of the SM can only hold a finite amount of information, it needs $O(\log n)$ cells to hold a counter value. When CPARSE$(q_0, 0, f, 0, x, S)$ is called, the input string x will eventually be divided into $O(n)$ substrings and the (parsing) problem will be split into $O(n)$ subproblems. For each subproblem, it takes at least $O(\log n)$ cells to store the initial and final counter values. Thus, an SM would need $O(n \log n)$ cells to execute CPARSE on input x. Since an SM can use only n cells for inputs of length n, it is impossible to implement the algorithm CPARSE on an SM. This problem can be resolved by relativizing the counter values as follows. Let $x_i=a_s...a_t$ be a substring of x and $m=t-s+1$. Suppose that the counter value before reading a_s (after reading a_t) is c_1 (c_2). If $c_1+c_2>m$, then the counter is always nonempty when the input head is reading a symbol of x_1. Thus, we have the following relativization process. Let c_1 and c_2 be the initial and final counter values of a subproblem. Without loss of generality, assume $|c_2-c_1|\leqslant m$. If $c_1+c_2\leqslant m+2$, then c_1 and c_2 should remain unchanged and do not have to be relativized. Otherwise, let $d=\lceil (c_1+c_2-m)/2\rceil$, $c_1=c_1-d+1$ and $c_2=c_2-d+1$. Clearly, the relativized counter values will be less than or equal to $m+1$ and can easily be stored using $O(m)$ cells. Note that now $b_1=\max\{0, \lceil (c_1+c_2-m)/2\rceil\}=1$ and $b_2=\lfloor (c_1+c_2+m)/2\rfloor=m+1$. The above discussion results in the following algorithm which can be implemented on an SM.

Algorithm PCPARSE$(q_1, c_1, q_2, c_2, x, S)$;
begin
{Let $x=a_1...a_n$. For simplicity, assume $n>0$.
b_1 and b_2 are defined the same as before}
1. {If $n=1$ then solve the problem directly}
if $n=1$ **then**
if $(q_1, a_1, \text{pos}(c_1))\rightarrow(q_2, c_2-c_1)\in\delta$ **then** $S:=(q_1, a_1, \text{pos}(c_1))\rightarrow(q_2, c_2-c_1)$
else $S:=$"no"
2. **else**
2.1. {Relativize counter values}
if $c_1+c_2>n+2$ **then**
$d:=\lceil (c_1+c_2-m)/2\rceil$; $c_1:=c_1-d+1$; $c_2:=c_2-d+1$;
2.2. {Split the problem}
$i:=\lceil n/2\rceil$;
row1 $:=$ lastrow$(E, q_1, c_1, a_1...a_i)$;
row2 $:=$ lastrow$(E^R, q_2, c_2, a_n...a_{i+1})$;
2.3. {Find a j such that $\exists p\in Q$, $p\in$row1$(j)\cap$row2(j)}
$j:=b_1$;
while row1$(j)\cap$row2$(j)=\emptyset$ **do**
$j:=j+1$
end;

if $j > b_2$ **then** $S :=$ "no"; **return**
else $p :=$ an arbitrary state in row1$(j) \cap$ row2(j);
3. {Solve simpler problems}
call PCPARSE$(q_1, c_1, p, j_1, a_1 \ldots a_i, S_1)$;
call PCPARSE$(p, j_2, q_2, c_2, a_{i+1} \ldots a_n, S_2)$;
4. $S := S_1, S_2$
endif
end.

Theorem 6.1. *Parsing of* 1-*NCA languages can be done on an OLIA in linear time.*

Proof. Let $x = a_1 \ldots a_n$ be an input string. We show that the algorithm PCPARSE can be carried out on an SM Z in $16n$ sweeps. As in the proof of Theorem 4.1, the worktape of Z is conceptually divided into four tracks. Each cell of the worktape is viewed as a 4×15 array of subcells. (Thus, the worktape of Z on input string x is a $4 \times 15n$ array of subcells.) Z operates in three stages: input stage, parse stage and output stage. The input stage takes n sweeps. In this stage, Z reads the input string x and stores it in the first n subcells of track 1. Meanwhile, Z also marks the middle of x. The parse stage takes $12n$ sweeps. In this stage, Z executes PCPARSE$(q_0, 0, f, 0, x, S)$. Again, when the algorithm PCPARSE divides a problem into two subproblems, Z splits the problem and solves the two subproblems in parallel. At the end of this stage, a parse of x (with respect to M) is stored on the worktape. The output stage takes $3n$ sweeps to output the parse obtained in the parse stage. The parse is output in reversed order. In what follows, we sketch the operations of Z in the parse stage.

Similar to the proof of the Theorem 4.1, Z keeps a sequence of active blocks on the worktape. The active blocks are shifted one subcell to the right every sweep. We describe the operations of Z in an active block. Let B be the active block for subproblem $(q_1, c_1, q_2, c_2, a_s \ldots a_t)$. Let $m = t - s + 1$. The length of B is $3m$ subcells. When the block is created, the string $a_s \ldots a_t$ is placed on track 1, the states q_1 and q_2 are placed on track 2, and the counter values c_1 and c_2 are stored on tracks 3 and 4 as unary strings 1^{c_1} and 1^{c_2}, respectively. The strings $a_s \ldots a_t$, 1^{c_1} and 1^{c_2} are adjusted to the left boundary of B. The middle of $a_s \ldots a_t$ (i.e. symbol a_i, where $i = \lceil (s+t)/2 \rceil$) is marked by a $\hat{}$, [see Fig. 9(a)]. From the relativization process, we know that $c_1, c_2 \leqslant 2m + 2$. If $m = 1$, Z solves the problem directly as in the step 1 of the algorithm PCPARSE and stores the parse in B. Suppose that $m > 1$. Then Z executes steps 2.1–2.3 in three phases. Without loss of generality, we assume $c_1 \geqslant c_2$. In the sequel, all the data movements and locations of subcells are taken to be with respect to block B.

In the first phase, Z reverses the substring $a_{i+1} \ldots a_t$, relativizes the counter values, and places the states q_1 and q_2 in the appropriate subcells so that the function lastrow can be computed. We describe the operations of Z in this phase. Let $d = c_1 - c_2$. Denote the substring of 1^{c_1} consisting of the last $d+1$ 1's by $[c_1, c_2]$. In the first $m/2$ sweeps, Z reverses the substring $a_{i+1} \ldots a_t$ and marks the middle of $[c_1, c_2]$ with a $\bar{}$ [see

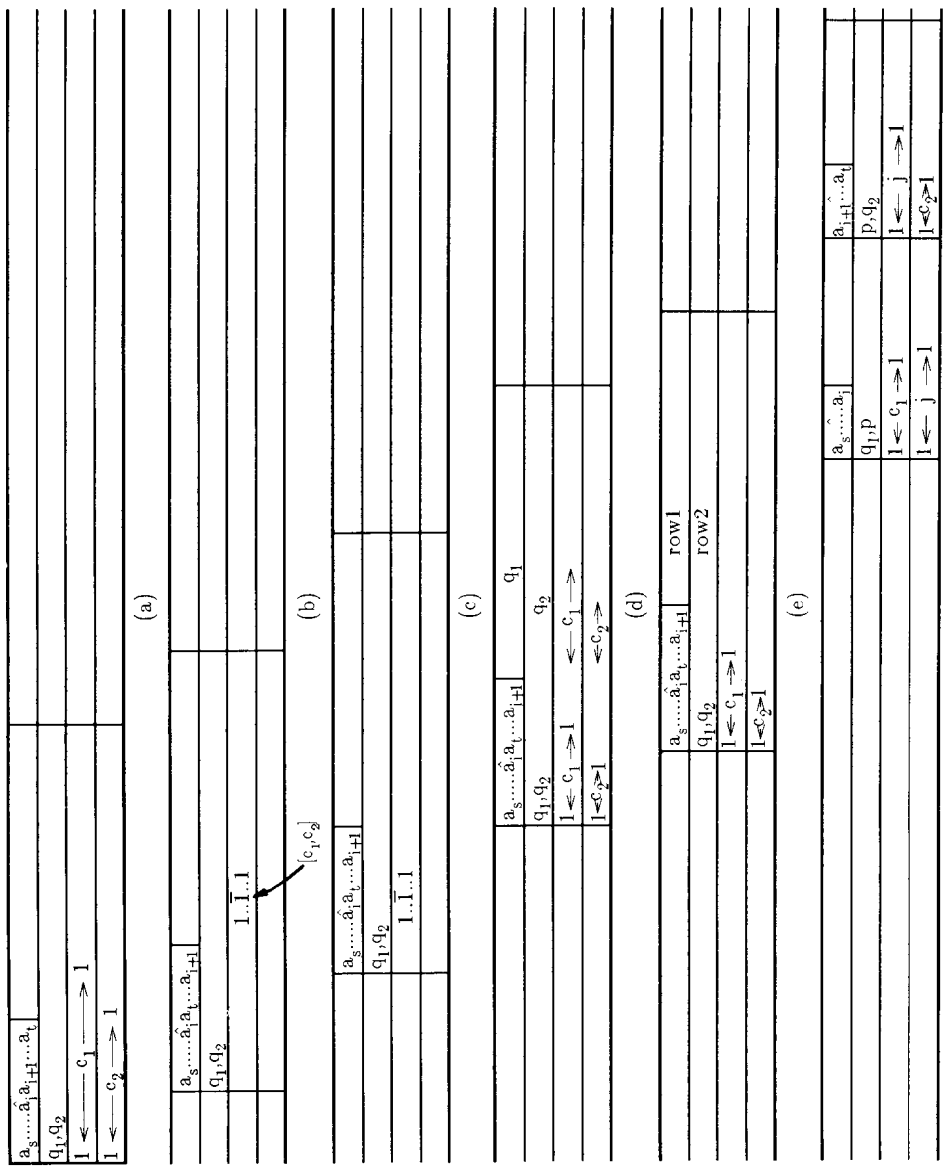

Fig. 9. The changes of block B's configuration.

Fig. 9(b)]. In the next $m/2$ sweep, Z determines whether or not $c_1+c_2 \leqslant m$. If the marker $\bar{}$ is not to the right of $\hat{}$(i.e. $c_1+c_2 \leqslant m+2$), then c_1 and c_2 do not need to be relativized. In this case, Z simply places the state q_1 in the $(m+c_1+1)$st subcell of track 1 and the state q_2 in the $(m+c_2+1)$st subcell of track 2. This can be done in m sweeps. If the marker $\bar{}$ is to the right of $\hat{}$(i.e. $c_1+c_2>m$), Z computes the relativized c_1 and c_2 as follows. It shifts the substring $[c_1, c_2]$ to the left at unit speed. When the marker $\bar{}$ is one subcell to the right of $\hat{}$ the last symbol of $[c_1, c_2]$ is at the $(\lceil m/2 \rceil + \lceil d/2 \rceil)$th subcell of track 3 [see Fig. 9(c)]. Note that now the (horizontal) distance between a_s and the last symbol of $[c_1, c_2]$ is equal to or one less than the relativized c_1, depending on the parity of m and d, and the distance between the last symbol of $[c_1, c_2]$ and a_{i+1} is one or two less than the relativized c_2, depending on the parity of m and d. It is easy to see that, using m more sweeps, the unary representations of the new c_1 and c_2 can be obtained and adjusted to the left boundary of B. Meanwhile, Z places the state q_1 in the $(m+c_1+1)$st subcell of track 1 and the state q_2 in the $(m+c_1+1)$st subcell of track 2. The configuration of B after the first phase is shown in Fig. 9(d). The total number of sweeps used in this phase is $7m/2$.

The second phase has $m/2$ sweeps. In the second phase, Z computes $\text{row1} = \text{lastrow}(E, q_1, c_1, a_s \ldots a_i)$ and $\text{row2} = \text{lastrow}(E^R, q_2, c_2, a_t \ldots a_{i+1})$ and stores them on tracks 1 and 2. In the computation, the $(m+j+1)$st subcell of track 1 (track 2) is used to hold the entries in column j of matrix E (E^R), $0 \leqslant j \leqslant m+1$. At the same time, Z reverses the substring $a_t \ldots a_{i+1}$ back to $a_{i+1} \ldots a_t$. Figure 9(e) shows the configuration of B at the end of the second phase.

In the third phase, Z first marks the entry $\text{row1}(j)$ with a special symbol, where j is the smallest k satisfying $\text{row1}(k) \cap \text{row2}(k) \neq \emptyset$. The block B is split into two blocks. The left block is of length $3\lceil m/2 \rceil$ and the right block is of length $3\lfloor m/2 \rfloor$. Z arbitrarily chooses a state p in $\text{row1}(j) \cap \text{row2}(j)$ and places it in both left and right blocks. This takes $2m$ sweeps. Meanwhile, Z stores the unary string 1^j on track 3 of the right block and shifts a copy of it to the left. The copy of 1^j will be stored on track 4 of the left block. Z also shifts the substring $a_{i+1} \ldots a_t$ and state q_2 into the right block. During this rearranging process, Z also marks the middles of $a_s \ldots a_i$ and $a_{i+1} \ldots a_t$. At the end of this stage, the problem $(q_1, c_1, q_2, c_2, a_s \ldots a_t)$ is divided into two subproblems $(q_1, c_1, p, j, a_s \ldots a_i)$ and $(p, j, q_2, c_2, a_{i+1} \ldots a_t)$. See Fig. 9(f).

It is easy to see that the parse stage takes $12n$ sweeps. Thus, Z operates in $16n$ sweeps. □

Corollary 6.2. *1-NCA languages can be parsed on a single-tape Turing machine in* $O(n^2)$ *time and* $O(n)$ *space.*

7. Conclusion

We have shown that the following problems can be solved on an OLIA in linear time: (1) parsing of discrete 2-TNFT transductions, (2) parsing of LCFL's, (3) parsing

of 1-NCA languages, and (4) string shuffling problem. For these problems, we were interested in not only the membership but also a parse. Since the nodes in an OLIA are identical finite-state machines and the communication between the nodes is one-way, the VLSI implementation of such an array is simple. In the paper, all the constructions were described in terms of an SM – the characterizing machine of an OLIA. Since an SM is a uniprocessor machine, we did not have to deal with problems such as concurrency and synchronization. This greatly simplified the presentation.

References

[1] K. Culik II, J. Gruska and A. Salomaa, Systolic trellis automata, *Internat. J. Comput. Math.* **16** (1984) 3–22.

[2] S. Ginsburg and G. Rose, The equivalence of stack counter accepters and quasi-realtime acceptors, *J. Comput. and System Sci.* **8** (1974) 243–269.

[3] S. Graham and M. Harrison, Parsing of general context-free languages, *Advances in Computers, Vol. 14* (Academic Press, NY, 1976) 77–185.

[4] S. Greibach, A note on the recognition of one counter languages, *RAIRO Inform. Théor.* **5** (1975) 5–12.

[5] D. Hirschberg, A linear space algorithm for computing maximal common sequences, *Comm. ACM* **18** (1975) 341–343.

[6] O. Ibarra and S. Kim, Characterizations and computational complexity of systolic trellis automata, *Theoret. Comput. Sci.* **29** (1984) 123–153.

[7] O. Ibarra, S. Kim and M. Palis, Designing systolic algorithms using sequential machines, *IEEE Trans. Comput.* **35** (1986) 531–542.

[8] O. Ibarra, M. Palis and J. Chang, On efficient recognition of transductions and relations, *Theoret. Comput. Sci.* **39** (1984) 89–106.

[9] O. Ibarra, M. Palis and S. Kim, Some results concerning linear iterative (systolic) arrays, *J. Parallel and Distributed Comput.* **2** (1985) 182–218.

[10] O. Ibarra and M. Palis, VLSI algorithms for solving recurrence equations and applications, *IEEE Trans. Acoust. Speech Signal Process.* **35** (1987) 1046–1064.

[11] H. Kung and M. Lam, Fault-tolerance and two level pipelining in VLSI systolic arrays, in: P. Penfield, Jr., ed., *Proc. 1984 Conference on Advanced Research in VLSI*, MIT, MA (1984) 74–83.

[12] C. Savage and M. Stallmann, Decomposability and fault-tolerance in one-dimensional array algorithms, in preparation.

[13] J. Van Leeuwen and M. Nivat, Efficient recognition of rational relations, *Inform. Process. Lett.* **14** (1982) 34–38.

[14] D. Younger, Recognition and parsing of context-free languages in time n^3, *Inform. and Control* **10** (1967) 189–208.